12/15/03

SOURCE CODE LISTING

```
/***************************************************************************
 *
 * File:                cgxovlay.c
 *
 * The CGX overlay operations, they implement all of the secure Kernel commands
 * and are internal to the secure Kernel, not to be shared with customer.
 *
 * Copyright (c) 1996, 1997, 1998 by IRE Secure Solutions, Inc.
 * All rights reserved.
 *
 *
 * REVISION HISTORY:
 *
 * 09-Sep-96 TFO: Created File
 * 20-Sep-96 TFO: Added code for CGX_GEN_KEY and CGX_GEN_KEK
 * 23-Sep-96 TFO: Added call to initialize the cryptoblk object
 * 23-Sep-96 TFS: Filled in hash_init and hash_data
 * 08-Nov-96 TFO: Removed the MAC commands, no longer used
 * 08-Nov-96 TFS: Hash commands only supported in PC environment.
 * 12-Nov-96 TFO: Removed: these cmds: CGX_LOAD_PUBKEY, CGX_UNLOAD_PUBKEY,
 *                CGX_WRAP_PUBKEY, CGX_DESTROY_PUBKEY, and CGX_UNWRAP_PUBKEY.
 * 04-Dec-96 TFO: Removed references to ATLAS/GDS via ifdefs
 * 05-Dec-96 TFO: Removed the restore member
 * 05-Dec-96 TFS: Changed hash operations to be byte-oriented.
 * 11-Dec-96 TFO: Added new operations: CGX_EXPORT_KEY, CGX_DERIVE_KEY, and
 *                CGX_TRANSFER_KEY.  Also, modified the operations:
 *                CGX_GEN_KEY, CGX_GEN_KEK, CGX_LOAD_KG, CGX_STREAM for
 *                the new trusted key heirarchy scheme.
 * 17-Dec-96 TFS: Made cover_key private, added import_key and provide
 *                keyed hash.
 * 24-Jan-97 TFO: Added GEN_PUBKEY, EXPORT_PUBKEY, SIGN and VERIFY commands
 * 27-Jan-97 TFS: Merged cgxovlay_gen_pubkey and cgxovlay_gen_newpubkey into
 *                    cgxovlay_pubkey.
 * 16-Feb-97 TFO: Added CGX_TRANSFORM_KEY command, removed test cmd, fixed
 *                random and port pubkey cmds, and create LSV hash.
 * 27-Feb-97 TFS: Changed pubkey operations and hash interfaces.
 * 04-Mar-97 TFO: Fixed CGX_TRANSFORM_KEY, didn't create the HMAC key
 *                correctly.
 * 11-Mar-97 TFO: Corrected buffer interfaces and replaced _flip with the
 *                buffer_flip operation.
 * 14-Mar-97 TFO: Added RC5
 * 24-Mar-97 TFO: Added call to seckey_load_kek in the uncover key case
 * 26-Mar-97 TFO: Changed it so secret keys remain in internal form unless
 *                they are exported.
 * 28-Mar-97 TFS: Added extended algorithms and changed use of digest in
 *                    hash_cntxt.
 * 07-Apr-97 TFO: Performed ROM optimizations, fixed export_key for RC5/HMAC
 * 24-Apr-97 TFO: Changed interface to the kcr_get_seckey calls to reflect
 *                new interface for extended keys.
 * 28-Apr-97 TFO: Removed kcr_used from cgxovlay_get_chipinfo
 * 30-Apr-97 TFS: Changes to support expanded key cache
 * 03-Jun-97 TFS: Changes based on finalized paged memory scheme.
 * 04-Jun-97 TFO: Moved global data to globals.c
 * 22-Aug-97  JS: Changed several mem_cpy's and memset's to correct
 *                lengths for target
 * 29-Aug-97 TFO: added new memory unit conversion intfc
 * 27-Oct-97 TFO: allow public keys to have lengths between 512 and 2048
*                bits with increments of 8 bits. Do this for the PC
*                target only, will solve 2181 later.
* 30-Oct-97 TFO: The chipinfo command will not return info correctly to
*                application in 2181 target mode.
* 07-Nov-97 TFO: Moved extende3d program operation internals to cgxovlyl.c
*                because kernel ROM 0 bank filled up.  Also performed
*                full rewrite of it.
* 10-Nov-97 TFO: As part of cgxovlay_initialize clear out any extended
*                program space that was previously allocated.
* 06-Dec-97 TFO: Added a private operation to bump ptr and datapage
* 23-Mar-98 TFO: fixed the operation, cgxovlay_hash_crypt, to be more like
*                HW block.  This means it allows valid offset and op_offset.
*                Later the full combine driver will be provided for faster
*                pipelined operations.
* 16-Apr-98 TFO: removed cis_init and pcdb_init, its done out of kernel_init
* 17-Apr-98 TFO: moved cgxovlay_load_kg and cgx_signature to
*                cgxovlyl.c, bank ran out space
*
*****************************************************************************/

/*****************************************************************************
* Include files.
*****************************************************************************/
#ifndef ADI2181
#include    <stdio.h>
#include    <stdlib.h>
#include    <io.h>
#include    <errno.h>

#undef NULL
#endif

#include    "std.h"
#include    "cgx.h"
#include    "random.h"
#include    "diag.h"
#include    "kernel.h"
#include    "tv.h"
#include    "crypcntx.h"
#include    "cgxovlay.h"
#include    "pcdb.h"
#include    "secretky.h"
#include    "seckey.h"
#include    "pagedmem.h"
#include    "kcr.h"
#include    "hash.h"
#include    "bignum.h"
#include    "pubkey.h"
#include    "dsa.h"
#include    "dh.h"
#include    "rsa.h"
#include    "buffer.h"
#include          "globals.h"
#include          "dpage.h"
#include          "kcs.h"

#if !defined(ADI2181)
```

```
UINT16  lsvl = 0x2d34;
#endif /* ADI2181 */

extern CGX_FUNC ExtendedEntry;

/************** CGXOVLAY DATA DECLARATIONS ************/

                  /* CGX Initialization Operations */

/*
 *    This operation is used to initialize anything specific to any of the
 *    _cgx_ operations.  This operation is invoked by the secure Kernel at
 *    bootup/reset.  It is called by the operation: kernel_initialize(), see
 *    kernel.c
 */
void
cgxovlay_initialize(void)
{
      kcr_memory_init();
      random_init();
      seckey_init();
      pubkey_init();
      ExtendedEntry = (CGX_FUNC)0xFFFF;
}

                  /* CGX Utility Operations */
/*
 *    General operation to create a secret key and copy it into a KCR
 *    register.  This has a slightly overload interface in order to be
 *    useable by several operations.  This is only done to conserve in
 *    ROM.
 *
 *IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE
 *IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE
 *IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE
 *IMPORTANT NCTE IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE
 *    All keys when they are created or loaded must be brought into the
 *    kernel via this operation, and this operation only.  This operation
 *    performs the necessary checks to validate the proper kernel key
 *    heirarchy.  Bypassing this operation could allow the kernel key
 *    heirarchy to be violated.
 *
 *    Furthermore, the operation _cgxovlay_cover_key must be the only
 *    operation used to cover a RED key and return it to the application,
 *    for the same reasons just explained.
 *
 *    Moreover the operation, _cgxovlay_uncover_key, must be the only
 *    operation used to uncover a BLACK secret key from the application
 *    into a RED one.
 *
 *    These rules must be enforced else the kernel key management scheme
 *    will go down the hopper.  Before making changes to the flow of keys
 *    please consult with one of these operations.
 */
UINT16
_cgxovlay_load_key(secretkey *sk, kcr r, UINT16 use, UINT16 k_type,
          UINT16 length, DPAGE bk_dp, secretkey *bk, DPAGE kek_dp, crypto_cntxt
*kek_cc, BOOL rkekFlag)
{
#if !defined(TARGETPATCH)
    seckey        *kcr_seckey;           /* seckey in a KCR              */
    seckey        kcr_sk;              /* memory for seckey in a KCR    */
    secretkey     local_k;               /* Scratch secretkey            */
    crypto_cntxt cc;                     /* Local copy                   */
    UINT16        kcr_type;              /* kcr's attr field             */
    UINT16        kcrtmp;                /* temp kcr attr type           */
    kcr           cckcr;                 /* Crypto Context KCR id        */
    UINT16        ccattr;                /* Cc KCR's attribute field     */
    UINT16        rc;                    /* Return value                 */
    UINT16        *dptr = (UINT16 *)&cc; /* Local pointer to cc          */

    if (rc = kcr_check(r, KCR_VALID_T|KCR_LSV_T)) {
        return rc;
      }

    /* Generate kcr_type from the use parameter (and, possibly, the
     * KEK. The result can then be validated.
     */

      /* use describes the KCR type ie: KCR_GKEK*/
    if (!use) {
        return CGX_BAD_KEY_TYPE_S;
      }

    if (kek_cc) {
        /* Copy user-specified arguments to working copies */
        mem_cpyDS(&knlDataPage, (MEMCPY_TYPE)&dptr, &kek_dp,
(MEMCPY_TYPE)&kek_cc,
                sizeof(crypto_cntxt));

        kek_cc = &cc;          /* Now, point to the local */

          /* return the KCR index 1-14  */
        cckcr = crypto_cntxt_kcr(kek_cc);
        if (rc = kcr_check(cckcr, KCR_VALID_T))
            return rc;

            /* get the attributes of the KCR in question */
        ccattr = kcr_get_attr(cckcr);
    }

    /* All keys in the system are untrusted (by default) with the
     * following exceptions:
       *            1. GEN_RKEK - a recovery KEK is always trusted.
     *        2. GEN_KEK - a generated KEK is always trusted, and
     *        3. GEN_KEY - can be either trusted or untrusted as
     *                long as parent is trusted. If parent is
     *                untrusted and a trusted key is requested,
     *                fail the request.
     */

    switch (knlCommand) {
     case CGX_SAVE_KEY:

          /* set the kcr_type */
          kcr_type = KCR_TRUSTED;

          /* then the kek_cc better be an RKEK */
          if (!kek_cc)
               return CGX_FAILED_SAVE_KEY_S;

          else /* get the key type based of the attribute bits and verify
KCR_RKEK */
               if( (kcr_get_key_type(ccattr)) != KCR_RKEK)
                    return CGX_FAILED_SAVE_KEY_S;

          break;

   case CGX_GEN_RKEK:          /* AN RKEK IS ALWAYS TRUSTED */
       kcr_type = KCR_TRUSTED;

         /*
          * Ensure that the only type of key one can create or load
          * with the GEN_RKEK operation is a KCR_RKEK.
          */

         if( (use & KCR_KTYPE_MASK) != KCR_RKEK )
            return CGX_INVALID_KEY_GEN_S;

       break;

   case CGX_GEN_KEK:
       kcr_type = KCR_TRUSTED;

     /*
      *    Ensure that the only type of key one can create or load
      *    with the GEN_KEK operation is a KCR_GKEK.
      */
     if( (use & KCR_KTYPE_MASK) != KCR_GKEK )
          return CGX_INVALID_KEY_GEN_S;

       break;

   case CGX_GEN_KEY:
       /* If the application requested an untrusted key, always
        * allow this under a GEN_KEY. Otherwise, inherit the
        * trust of the parent key. By default, a generated key
        * should be trusted. But if the key is generated under an
        * untrusted key, inherit the trust of the parent.
      *
      *IMPORTANT IMPORTANT IMPORTANT:
      *    The more subtle point about this code is that if one was
      *    to use the GEN_KEY operation and you specify that it
      *    be TRUSTED, BTW that is the default mode, the following
      *    code will only assign the parent or the KEK's trust level.
      *    This means that if you were trying to place a GEN_KEY that
      *    was to be trusted under a untrusted KEK or parent the
      *    GEN_KEY or this code makes it untrusted.  Therefore, the
      *    GEN_KEY operation can be thought of a dumb operation in that
```

```
         *    it will ignore the trusted requests and let the parent
         *    define it.  We say this in the API so user beware, we don't
         * fail but decided to implement the feature this way.  Then
         * the key tree the application is building will
         *    define the trust levels.
          */
        if ((kcr_type = (use & KCR_TRUST_MASK)) != KCR_UNTRUSTED) {
            /* Inherit the trust of the parent key. */
            if (kek_cc)
                kcr_type = kcr_get_trust(ccattr);
            else
                /* Dangling keys are untrusted */
                kcr_type = KCR_UNTRUSTED;
        }

      /*
        *    The GEN_KEY operation can generate any type of key but a
        *    LSV, GKEK, or RKEK.  If it tries fail it.  In fact we
        *    could allow it to generate the GKEK if the application
        *    supplied a crypto_cntxt to the LSV, in fact it would work
        *    but lets now enforce the key generation routes from the
        *    right key.
        *
        *    The LSV check is really redundant because of the code
        *    below this checks if some fool is trying to create an
        *    LSV.  But it doesn't cost anything so we might as well
        *    do it here, makes things consisted, readability wise.
        */
      if( use & (KCR_LSV|KCR_GKEK|KCR_RKEK) )
            return CGX_INVALID_KEY_GEN_S;

        break;

    default:
        kcr_type = KCR_UNTRUSTED;

      /*
        *    The DEFAULT operations can generate any type of key but a
        *    LSV, GKEK, or RKEK.  If it tries fail it.
        *
        *    The LSV check is really redundant because of the code
        *    below this checks if some fool is trying to create an
        *    LSV.  But it doesn't cost anything so we might as well
        *    do it here, makes things consisted, readability wise.
        */
          if( use & (KCR_LSV|KCR_GKEK|KCR_RKEK) )
            return CGX_INVALID_KEY_GEN_S;

        break;
    }

    /* Validate kcr type passed in use. The application must
     * specify something! Futher, only single bits (currently,
     * only RKEK, GKEK, KEK, KKEK, HMAC or K) can be requested.
     */
    kcrtmp = use & KCR_KTYPE_MASK;
    if (kcrtmp & (KCR_HMAC|KCR_K|KCR_KKEK|KCR_RKEK|KCR_KEK|KCR_GKEK))
```

```
        kcr_type |= kcrtmp;
    else
        return CGX_KEK_REQUIRED_S;

    /* if user requests a CGX_HMAC_A type of secret key we must force */
    /* the key type kcr attribute to KCR_HMAC, don't allow it to be */
    /* defined as a KEK, GKEK, RKEK or K.  Also, if via the GEN_KEK cmd */
    /* don't mess with type because it should of come in right if it came */
    /* from the GEN_KEK cmd, otherwise app is trying to spoof. */
    if( (knlCommand != CGX_GEN_KEK) && (knlCommand != CGX_GEN_RKEK) && (k_type
== CGX_HMAC_A) ) {
        kcr_type = ((kcr_type & ~KCR_KTYPE_MASK) | KCR_HMAC);
    }

    /* Note parent's trust attribute in the attribute bits */
    if (kek_cc)
    {
        if (kcr_is_untrusted(ccattr))
            kcr_type |= KCR_PARENT_UNTRUSTED;
    }
    else if ((knlCommand != CGX_GEN_KEK) && (knlCommand != CGX_GEN_RKEK))
                /* No parent KEK, use untrusted */
        kcr_type |= KCR_PARENT_UNTRUSTED;

    /* Check if valid type of key requested */
    if ((rc = kcr_validate(kcr_type, k_type, &length)) != CGX_SUCCESS_S)
        return rc;

    /* If passed secret key object is NULL, generate a key for client */
    if (sk == (secretkey *)NULL)
    {
        /* Initialize the scratch key to all zero's */
        secretkey_init(&local_k);

        /* Create actual user key into a local scratch key */
        if (rc = secretkey_gen_key(&local_k, k_type, length))
            return rc; /* Oops, something failed, bail */

        sk = &local_k; /* Establish a pointer to it */
    }

    /* Get copy of secretkey (sk) in kcr's seckey */
    kcr_seckey = kcr_get_seckey(r, (seckey *)&kcr_sk);
    seckey_secretkey2seckey(kcr_seckey, sk);
    seckey_set_length(kcr_seckey, length);
    seckey_set_type(kcr_seckey, k_type);

    /* mark the KCR to contain a newly generated user key */
    kcr_set_key_attr(r, kcr_type);

    /* Now transform the key into internal representation */
    seckey_setup(kcr_seckey, TRUE);

      /* place the secret key into the kcr_index referred to as r */
    kcr_put_seckey(r, (seckey *)&kcr_sk);

    /* If requested (i.e., bk has a value, cover the key using the
      * specified kek).
      */

     if (bk) {
         rc = _cgxovlay_cover_key(r, bk_dp, bk, kek_cc, rkekFlag);
         if (rc != CGX_SUCCESS_S) {
             kcr_destroy(r);
             return rc;
         }
     }

     /* Ask the crypto-block to remove its installed key if dest_kcr is KG */
     seckey_remove_loaded(r);

     if (kcrtmp == KCR_KKEK) {
         seckey_load_kek(kcr_seckey);
     }

     return rc;
#else
     return CGX_FAIL_S;
#endif
}
/*
 *    This operation returns a seckey object if the key is fit to be
 *    used in traffic encryption.  This will fail if a LSV, GKEK, RKEK or
 *    a longer key than export allows is used.  Will also fail if an
 *    empty KCR or invalid KCR is specified.
 *
 *    It reads back the actual secretkey red bits into the arg, sk,
 *    because the caller can't simply refer to the ptr verion of the
 *    secretkey.  This is because the actual secret key could be sitting
 *    in an entirely different data RAM page.
 */
seckey *
_cgxovlay_valid_crypto_key(UINT16 kcr_key, seckey *sk)
{
      /* get the KCR secret key */
      if( kcr_check(kcr_key, (KCR_HMAC_T|KCR_ANY_KEK_T|KCR_VALID_T|KCR_EMPTY_T))
)
             return (seckey *)NULL; /* bad KCR key to use for crypto operations
*/

      return kcr_get_seckey(kcr_key, sk);
}

/*
 *    This is the low level operation to call to perform the
 *    crypto operation decrypt or encrypt for the commands:
 *    CGX_HASH_ENCRYPT, CGX_HASH_DECRYPT, CGX_ENCRYPT, and
 *    CGX_DECRYPT ONLY.  Do not use these commands for any other
 *    operation or you will be shot on the spot.  This operation
 *    assumes the key already or to be loaded will be a data key.
 *    direction: 0 decrypt
 *               non0 encrypt
 */
UINT16
```

```
_cgxovlay_crypto(crypto_cntxt *cc, DPAGE dest_dp, UINT16 *dest_p, DPAGE src_dp,
UINT16 *src_p, UINT16 byte_cnt, UINT16 direction, UINT16 pad)
{
      UINT16            kcr_key;     /* KCR id of key loaded or to load */
      seckey            sk;    /* seckey object */

      /* obtain the kcr location of the secret key to be used */
      kcr_key = _cgxovlay_kcr(cc);

      /* check if a valid KCR key can be used for crypto like operations */
      if( _cgxovlay_valid_crypto_key(kcr_key, (seckey *)&sk) == (seckey *)NULL )
            return CGX_INVALID_REG_S; /* no such KCR key found */

      /* encrypt or decrypt the data */
      seckey_encrypt_decrypt((seckey *)&sk, cc, direction, dest_dp,
                        (UINT16 *)dest_p, src_dp, (UINT16 *)src_p,
                        (UINT16)byte_cnt, pad);

      return CGX_SUCCESS_S;
}

/*
 *    Based on a black secretkey and the cc or kek, uncover the black
 *    secretkey only if the kek is not trusted.  If all goes well
 *    pass a pointer to the seckey object in the scratch KCR location.
 */
seckey *
_cgxovlay_scratch_key(DPAGE bk_dp, secretkey *bk, DPAGE kek_dp, crypto_cntxt
*kek_cc, seckey *sk)
{
      crypto_cntxt cc;
      UINT16        *dptr = (UINT16 *)&cc;
      UINT16        *kek_ccp = (UINT16 *)kek_cc;

      /* First get a copy of the black key's cc to check the kek */
      mem_cpyDS(&knlDataPage, (MEMCPY_TYPE)&dptr, &kek_dp,
(MEMCPY_TYPE)&kek_ccp,
            sizeof(crypto_cntxt));

      /* ensure valid KCR number */
      if(kcr_check(crypto_cntxt_kcr(&cc), KCR_VALID_T|KCR_EMPTY_T))
            return (seckey *)NULL;

      /* check the black key to move's kek to see if it is a trusted */
      /* kek.  If it is trusted the black can not be moved, you can */
      /* only move a key from a untrusted branch to any branch */

      if( kcr_is_trusted(kcr_get_attr(crypto_cntxt_kcr(&cc))) )
            return (seckey *)NULL;

      /* First uncover the black secret key using the application supplied */
      /* black secretkey(a1) and kek(a2) */
      /* place the uncovered key into the scratch KCR location */
      if( _cgxovlay_uncover_key(KCR_SCRATCH_N, bk_dp, (secretkey *)bk,
                        kek_dp, (crypto_cntxt *)kek_cc) != CGX_SUCCESS_S )
            return (seckey *)NULL;
```

```
        /* Next get a pointer to the red internal secret key, the */
        /* application's uncovered black secretkey. */
        return kcr_get_seckey(KCR_SCRATCH_N, sk);
}

/*
 *      This operation returns the KCR number based on the config bits
 *      of the crypto_cntxt.
 */
UINT16
_cgxovlay_kcr(crypto_cntxt *cc)
{
        /* check the type of config mode, if NOLOAD refer to the key  */
        /* already loaded into the crypto-block */
        if( crypto_cntxt_crypto(cc) & CGX_NOLOAD_C )
                return seckey_key_id_loaded(CGX_DES_A);
        else
                return crypto_cntxt_kcr(cc);
}

/*
 *      general operation that cleans the scratch register out and returns
 *      the code passed in.
 */
UINT16
_cgxovlay_cleanup(UINT16 rc)
{
        /* clean up the scratch KCR so no one can use the result */
        kcr_destroy(KCR_SCRATCH_N); /* get rid of temporary KCR */

        return rc;
}

                        /* CGX Overlay Operations */
/*
 *      The CGX overlay arguments have access to the two kernel block
 *      pointers to access the command and status blocks.  To access the
 *      command block a pointer knlCmdBlock is provided and setup by the
 *      secure Kernel.  To access the status block the pointer knlStatusBlock
 *      is provided and setup by the secure Kernel. Furthermore, several
 *      macros are provided to get at the arguments to the command block. In
 *      fact at this time there is a maximum of 10 arguments.  Therefore, there
 *      is 10 of these macros to access each argument:
 *                      argument_1 ... argument_10
 *
 *      The CGX overlay operations can use them or access the cmdblock arguments
 *      directly via knlCmdBlock.  The macros are defined in kernel.h.
 */

/*
 * CGX_INIT
 *
 */
UINT16
cgxovlay_init(void)
{
        return _cgxovlay_init();
```

```
}

/*
 * CGX_DEFAULT
 *
 */
UINT16
cgxovlay_default(void)
{
      return _cgxovlay_default();
}

/*
 * CGX_RANDOM
 *
 */
UINT16
cgxovlay_random(void)
{
      random_dp(dpage_2, (UINT16 *)argument_2, (UINT16)argument_1);

      return CGX_SUCCESS_S;
}

/*
 * CGX_GET_CHIPINFO
 *
 */
UINT16
cgxovlay_get_chipinfo(void)
{
      return _cgxovlay_get_chipinfo();
}

            /* Encryption Commands */
/*
 * CGX_COVER_KEY
 *
 * Description: Cover srckcr using the crypto_cntxt and store
 * the result in the user's blkkey.
 */
/*
 *IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE
 *IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE
 *IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE
 *IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE
 *    All keys when they are created or loaded must be brought into the
 *    kernel via this operation, and this operation only.  This operation
 *    performs the necessary checks to validate the proper kernel key
 *    heirarchy.  Bypassing this operation could allow the kernel key
 *    heirarchy to be violated.
 *
 *    Furthermore, the operation _cgxovlay_cover_key must be the only
 *    operation used to cover a RED key and return it to the application,
 *    for the same reasons just explained.
 *
 *    Moreover the operation, _cgxovlay_uncover_key, must be the only
 *    operation used to uncover a BLACK secret key from the application
 *    into a RED one.
 *
 *    These rules must be enforced else the kernel key management scheme
 *    will go down the hopper.  Before making changes to the flow of keys
 *    please consult with one of these operations.
 */
UINT16
_cgxovlay_cover_key(kcr srckcr, DPAGE dbk_dp, secretkey *dest_bk, crypto_cntxt
*kek_cc, BOOL rkekFlag)
{
    UINT16        rc;           /* General return code              */
    UINT16        cntxt_type;   /* Type of the crypto context kcr   */
    UINT16        src_type;     /* Type of the kcr to be covered    */
    UINT16        attr;         /* Attribute field of srckcr        */
    secretkey     bk;           /* Local copy of blkkey             */
    crypto_cntxt  cc;           /* Local copy of crypto context     */
    kcr           cckcr;        /* Crypto context's kcr             */
    seckey        sseckey;      /* Local copy of source seckey      */
    UINT16        *sptr;

    /* get the type of the key and the kek */
    src_type = kcr_get_key_type(kcr_get_attr(srckcr));

    /* If a kek_cc was provided (i.e., non-NULL), fine we'll use it
     * below. If the kek_cc was NULL, then we had better be covering
     * a GKEK or an RKEK. If this is so, set up a local crypto context (to
     * reference the LSV) and proceed.
     */

    /* if NULL */
    if (!kek_cc)
    {
        if ( src_type & (KCR_GKEK|KCR_RKEK) )
        {
            /* Called with NULL and the type is a GKEK or an RKEK, therefore the
             * covering key is the LSV. Make it so...
             */

            kek_cc = &cc;
            crypto_cntxt_set_kcr(kek_cc, 0);    /* setting the kek_cc->key equal
to KCR zero */
        }
        else
            return CGX_KEK_REQUIRED_S;
    }

    cckcr = crypto_cntxt_kcr(kek_cc);  /* Assign the Kek KCR index to cckcr */


    /* Check to see that kcrs are valid and not empty... In the case of
     * the srckcr, make sure the app is not trying to cover the LSV. Also,
     * don't allow the crypto context kcr be the same as the source.
     */

    if (srckcr == cckcr)
        return CGX_KCR_SAME_S;
    if (rc = kcr_check(srckcr, KCR_VALID_T | KCR_EMPTY_T | KCR_LSV_T))
        return rc;

    if (rc = kcr_check(cckcr, KCR_VALID_T | KCR_EMPTY_T))
        return rc;

    cntxt_type = kcr_get_key_type(kcr_get_attr(cckcr));
      /* next. Check the rkekFlag to see if we are saving a key under an rkek
(TRUE) or
       * not (FALSE)
       */

      if(   !rkekFlag && (src_type == KCR_RKEK) )
            return CGX_INVALID_REG_S;

    /* Check for correct combinations of crypto_cntxt and RED kcr types. */

    switch (cntxt_type)
    {
    case KCR_LSV:
              /* The only thing that can be covered under the LSV is a GKEK or.
an RKEK. */
        if (src_type < KCR_GKEK)
            return CGX_INVALID_REG_S;
        break;

            case KCR_RKEK:
                /* Cannot use an RKEK to cover an LSV */
                if (src_type > KCR_GKEK)
                    return CGX_INVALID_REG_S;


                    /* if we are covering a key with an rkek then we better have
                     * come in from cgx_save_key. Else fail */
                    if(knlCommand != CGX_SAVE_KEY)
                          return CGX_INVALID_REG_S;



                    break;

    case KCR_GKEK:
    case KCR_KEK:
        /*
         * We must be covering something at level 3 (or below in the
         * case of KEK) in the key hierarchy (i.e. KEK, K, or HMAC).
         * Src_type can't be KCR_EMPTY - this was checked above...
         */
        if (src_type > KCR_KEK)
            return CGX_INVALID_REG_S;
        break;

    case KCR_KKEK:
        /* When covering with KKEK, assure that only data keys can
         * be covered by KKEK.
         */
    if (src_type != KCR_K)
        return CGX_INVALID_REG_S;
    break;

/*
 * If it wasn't a KEK of some type then fail, you can obly cover keys
 * with a KEK.
 */
default:
    return CGX_INVALID_REG_S;
}/* end switch */

/* For the LSV and GKEKs, setup the IV since we'll need to use
 * the fixed IV.
 */

if (cntxt_type >= KCR_GKEK) {
    crypto_cntxt_fix_iv(kek_cc);
} else {
    /*
    * Always force CBC mode regardless of CC KCR type.  Always force
    * KEK to be reloaded, pay a little more IO penalty for it.
    */
    crypto_cntxt_set_crypto(kek_cc, CGX_CBC_M|CGX_FORCELOAD_C);
}

kcr_get_seckey(srckcr, (seckey *)&sseckey);
attr    = kcr_get_attr(srckcr);

/*
 * No longer required because the secret key is left in the internal
 * form at all times except when it is exported.  This is to allow the
 * application to load DES keys directly into the external HW crypto
 * block in the black form.  If we left it in external form HW would
 * have to key weaken it.
 *
seckey_unsetup(&sseckey);
 *
 */

/* Copy key and it's attributes (length, extra and type) */
seckey_2secretkey(&sseckey, &bk);

        /*
   * Encrypt the key's length for later sanity check.
   */
   sptr = (UINT16 *)secretkey_key(&bk);
  sptr[CGX_SECRET_KEY_KLEN] = secretkey_length(&bk);

if ((rc = kcr_add_salt(secretkey_key(&bk), &attr)) == CGX_SUCCESS_S) {
    /* Encrypt uses 64-bit blocks, so we want to encrypt 7
     * 64-bit blocks (4 for the key + 3 for the hash). The
     * interface requires a byte count, so multiply by 8.
     */

    seckey_encrypt(kcr_get_seckey(cckcr, (seckey *)&sseckey), kek_cc,
               knlDataPage, (UINT16 *)secretkey_key(&bk),
                    knlDataPage, (UINT16 *)secretkey_key(&bk),
                CGX_RAW_SECRET_KEY_HASH_BYTE_LENGTH, CGX_ZERO_PAD);

        /* copy the black key back out to the application */
        sptr = (UINT16 *)&bk;
        mem_cpyDS(&dbk_dp, (MEMCPY_TYPE)&dest_bk, &knlDataPage,
(MEMCPY_TYPE)&sptr, sizeof(secretkey));
    }

    /* Clean up */
#ifndef ADI2181
    memsett(&bk, 0, sizeof(secretkey)); /* OK */
#endif
    seckey_remove_loaded(cckcr);

    return rc;
}

/*
 *IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE
 *IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE
 *IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE
 *IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE IMPORTANT NOTE
 *    All keys when they are created or loaded must be brought into the
 *    kernel via this operation, and this operation only.  This operation
 *    performs the necessary checks to validate the proper kernel key
 *    heirarchy.  Bypassing this operation could allow the kernel key
 *    heirarchy to be violated.
 *
 *    Furthermore, the operation _cgxovlay_cover_key must be the only
 *    operation used to cover a RED key and return it to the application,
 *    for the same reasons just explained.
 *
 *    Moreover the operation, _cgxovlay_uncover_key, must be the only
 *    operation used to uncover a BLACK secret key from the application
 *    into a RED one.
 *
 *    These rules must be enforced else the kernel key management scheme
 *    will go down the hopper.  Before making changes to the flow of keys
 *    please consult with one of these operations.
 */
UINT16
_cgxovlay_uncover_key(kcr destkcr, DPAGE bk_dp, secretkey *black, DPAGE kek_dp,
crypto_cntxt *kek_cc)
{
    UINT16          cntxt_type;   /* Type of the crypto context kcr     */
    UINT16          ktype;        /* Type of the uncovered key          */
    UINT16          k_type;       /* Key type of the uncovered key      */
    UINT16          rc;           /* General purpose return value       */
    UINT16          attr;         /* Key's attribute value              */
    UINT16          ccattr;       /* CC Key's attribute value           */
    secretkey       bk;           /* Local copy of blkkey               */
    crypto_cntxt    cc;           /* Local copy of crypto context       */
    kcr             cckcr;        /* Crypto context kcr                 */
    seckey          dseckey;      /* memory for a seckey                */
    UINT16          *dptr;
    UINT16          *sptr;
    /* Copy user-specified data to working copies */
    if (!kek_cc)
        return CGX_KEK_REQUIRED_S;

    /* mem copy in the kek_cc from somewhere in memory to the kernel memory and
name it &cc */
    sptr = (UINT16 *)kek_cc;
    dptr = (UINT16 *)&cc;
    mem_cpyDS(&knlDataPage, (MEMCPY_TYPE)&dptr, &kek_dp, (MEMCPY_TYPE)&sptr,
sizeof(crypto_cntxt));

    /* mem copy in black from somewhere in memory to the kernel memory and name
it &bk */
    sptr = (UINT16 *)black;
    dptr = (UINT16 *)&bk;
    mem_cpyDS(&knlDataPage, (MEMCPY_TYPE)&dptr, &bk_dp, (MEMCPY_TYPE)&sptr,
sizeof(secretkey));


    /* set cckcr to contain the KEK's KCR location */
    cckcr = crypto_cntxt_kcr(&cc);

    /* Verify that kcrs are valid and that the destination location is
     * not the LSV (can't overwrite LSV). In addition, make sure that the
     * crypto context is not empty (but can be LSV for GKEK or RKEK).
     */

    if (rc = kcr_check(destkcr, KCR_VALID_T | KCR_LSV_T))
        return rc;

    if (rc = kcr_check(cckcr, KCR_VALID_T | KCR_EMPTY_T))
        return rc;

      /* get the attributes of the KEK located in the KCR */
    ccattr = kcr_get_attr(cckcr);
    cntxt_type = kcr_get_key_type(ccattr); /* the kek type */

    /* Always force CBC mode, regardless of CC KCR type */
    crypto_cntxt_set_crypto(&cc, CGX_CBC_M|CGX_FORCELOAD_C);

    /*
     * if (dest_type != KCR_EMPTY && knlLoginId < CGX_OPERATOR_L)
     *    return CGX_PRIVILEGE_DENIED_S;    // Insufficient privilege
     */

    /* Assume that the crypto_cntxt contains a valid kcr. This must
     * be so, since we wouldn't allow the kcr to be populated if the
     * combination of parameters wasn't valid.
     */

              /* KEK type */
    switch(cntxt_type) {
    case KCR_LSV:
            /* Type can be an RKEK or a GKEK */
            /* The bk is either a GKEK or an RKEK and must be a tdes */
            if (secretkey_type(&bk) != CGX_TRIPLE_DES_A)
```

```
                return CGX_BAD_MODE_S;
            else
            ktype = KCR_GKEK | KCR_RKEK;
          /* fix the IV */
        crypto_cntxt_fix_iv(&cc);

        break;


    case KCR_RKEK:

            /* We do not support key restore under an RKEK. Therefore, if the
KEK is the RKEK, then FAIL !!! */

        return CGX_RKEK_UNCOVER_FAIL_S;
        break;
    case KCR_GKEK:
            /* Type can be KEK, KKEK, or K, otherwise, validate would have
             * failed. ktype will be checked after the key has been
             * uncovered....
             */

            ktype = KCR_KEK | KCR_KKEK | KCR_K;

      /* fix the IV */
        crypto_cntxt_fix_iv(&cc);

        break;

    case KCR_KEK:
        /* Type must be KEK, KKEK or K otherwise validate would have
         * failed. Use the user-provided IV - must be uncovering with
         * a level 3 (or lower, level 4, level 5 ...) key.
         */
        ktype = KCR_KEK | KCR_KKEK| KCR_K;
        break;

    case KCR_KKEK:
        /* Only valid type allowed to be stored under a KKEK is a K */
        ktype = KCR_K;
        break;

    default:
        /* Crypto context must contain a KEK! */
        return CGX_KEK_REQUIRED_S;
    }

    /* if key is an HMAC key then add that to the type */
    /* or the kcr_validate check will fail, will be validated */
    /* again below */

    /* looking at the algorithm */
      /* if a CGX_HMAC_A (algorithm) */
    if( (k_type = secretkey_type(&bk)) == CGX_HMAC_A )
        ktype |= KCR_HMAC;    /* set the KCR type */

    /* Validate the contents of the kcr for the black key */
    /*                                  KCR type,  algorithm type,      length of
key                    */
    if (rc = kcr_validate(ktype, secretkey_type(&bk), &secretkey_length(&bk)))
        return rc;

    /* Interface to the decrypt uses 64-bit blocks, so encrypt 7 64-bit
     * blocks (4 for the key + 3 for the hash value).
     */

      /* decrypt the black key with the KEK */
      /* first kcr_get_seckey() gets the actual KEK secret key material and
assign it to &dseckey */
    seckey_decrypt(kcr_get_seckey(crypto_cntxt_kcr(&cc), (seckey *)&dseckey),
&cc,
                knlDataPage, (UINT16 *)secretkey_key(&bk),
                knlDataPage, (UINT16 *)secretkey_key(&bk),
                CGX_RAW_SECRET_KEY_HASH_BYTE_LENGTH, CGX_ZERO_PAD);

      /* Remove the salt and obtain the attribute bits of the red key */
    if ((rc = kcr_remove_salt(secretkey_key(&bk), &attr)) == CGX_SUCCESS_S) {
        /* Initialize seckey we'll populate it with the key recreated
         * above and with the type, extra and length fields that were
         * part of the original key.
         */
        /* dseckey = kcr_get_seckey(destkcr, (seckey *)&dseckey); */
        seckey_secretkey2seckey(&dseckey, &bk);        /*  mem copies SRC red
key to dseckey */
        seckey_set_type(&dseckey, k_type);                        /* sets the A type
*/
        seckey_setup(&dseckey, FALSE);                            /* weakens key if
TRUE */
        kcr_put_seckey(destkcr, (seckey *)&dseckey);   /* put into destkcr */

        /* Now based on the real key attributes we need to re-check */
        /* using kcr_validate.  This is done here to validate for */
        /* HMAC keys, but will work for the general keys as well. */
        /* Also, the length is ignored, already adjusted in the */
        /* first call to kcr_validate */
        if (rc = kcr_validate(attr, k_type, &secretkey_length(&bk)))
            return rc;

            /* check the attributes against the ktype. If something isn't set,
then fail */
        if ( !(ktype & attr) )
                return CGX_BAD_KEY_ATTRIBUTES_S;

      /*
       *    Confirm the length of key is valid, must match the one
       *    stored outside of encrypted material.
       */
      sptr = (UINT16 *)secretkey_key(&bk);
      if( sptr[CGX_SECRET_KEY_KLEN] != secretkey_length(&bk) )
            return CGX_INVALID_LEN_S;

      /* If ktype is not an rkek of a gkek then, bring in the existing trust
       * mask, else, set to trusted */
        if ( !(ktype & (KCR_GKEK | KCR_RKEK) ) )
```

```
            attr &= (ktype | KCR_TRUST_MASK);
        else
            attr = (ktype & attr) | KCR_TRUSTED;

        /* Note parent's trust attribute in the attribute */
            if (kcr_is_untrusted(ccattr))
                attr |= KCR_PARENT_UNTRUSTED;

        kcr_set_key_attr(destkcr, attr);

        /* install the HW KKEK if a KKEK is uncovered */
        if (attr & KCR_KKEK) {
            seckey_load_kek(&dseckey);
        }
    }

    /* Clean up */
#ifndef ADI2181
    memsett(&cc, 0, sizeof(crypto_cntxt)); /* OK */
    memsett(&bk, 0, sizeof(secretkey)); /* OK */
#endif
    seckey_remove_loaded(destkcr);

    return rc;
}

/*
 * CGX_UNCOVER_KEY
 *
 * Description: Uncover a black key (argument_2) using the crypto_cntxt
 *          (argument_3) and store the result in the destination kcr
 *          (argument_1).
 */
UINT16
cgxovlay_uncover_key(void)
{
      return _cgxovlay_uncover_key((kcr) argument_1, dpage_2, (secretkey
*)argument_2,
                  dpage_3, (crypto_cntxt *) argument_3);
}

/*
 * CGX_GEN_KEK
 *
 *    This operation is responsible for creating a RED GKEK and
 *    placing it in the user desired KCR. A GKEK is by requirements
 *    a triple length DES key.
 */
UINT16
cgxovlay_gen_kek(void)
{
      /* create the secret GKEK and store it in the desired KCR */
      return _cgxovlay_load_key((secretkey *)NULL, (kcr)argument_1,
                           KCR_GKEK, CGX_TRIPLE_DES_A, 21, dpage_2,
                       (secretkey *)argument_2, knlDataPage, (crypto_cntxt
*)NULL, FALSE);
}
```

```
/*
 * CGX_GEN_RKEK
 *
 *    This operation is responsible for creating a RKEK and
 *    placing it in the user desired KCR. A RKEK is by requirements
 *    a triple length TDES key. An rkek can only be generated once
 *  a token has been verified. The args are as follows:
 *
 *    argument_1:       token The token
 *    argument_2:       kcr         The key cache reg.  Passed in by value.
 *    argument_3:       dhpk  The local dh covered private key.
 *    argument_4:       dhkek The local dh kek.  Must not allow for null kek.
 *    argument_5:       rkek  The black rkek to be returned
 *
 */

UINT16
cgxovlay_gen_rkek(void)
{

    UINT16          rc = CGX_SUCCESS_S;
      UINT16                  writeOverSn=TRUE;  /* This flag is used to cause
the token verify routine
                                                      * to automatically
(blindly) write over the tokens s/n
                                                      * with the chip's
s/n, providing that the flag is TRUE.
                                                      */

      UINT16            g_xlength;
      UINT16            *g_xdata;         /* create two variables to contain the
"remote" g^x
                                      * data and length from the token */

      /* reset the heap */
      pubkey_reset_heap();


      /* sanity check:
       * if the dhpk; generator/modulus/private_key or the rkek are NULL
       * then return CGX_FAILED_RKEK_S.  The dhpk can NOT be allowed to be red.
       */

       if ( argument_3==NULL || argument_4==NULL || argument_5==NULL)
            return CGX_FAILED_GEN_RKEK_S;



      /* If the token_verify routine returns SUCCESS, then gen rkek. */

      /*
       * If the token is ok, the g_xdata pointer will contain the g^y data from
the token.
       * The application passed in the generator, modulus and the covered
private x (to create g^xy).
```

```
       * So call a custom function which will call bigpow and generate g^xy, the
rkek. Once the rkek
       * is generated call cgxovlay_load_key to load the rkek and return the
covered rkek to the application
       */


      if( (rc=_cgxovlay_token_verify( dpage_1, (token_no_data *)argument_1,
                        writeOverSn, (UINT16 *)&g_xlength, (UINT16 **)&g_xdata )
)== CGX_SUCCESS_S)
      {
#if defined(SIM2181) || defined(VSIM2181)
            goto RkekDone;
#endif
            /* Token has been verified.  The token data contains the public
value: g^y.
             * next, call a function to call bigpow and complete the
exponentiation
             */

            if ((_cgxovlay_complete_negkey((BigInt *)NULL, dpage_3, argument_3,
dpage_4, argument_4,
                        (UINT16)g_xlength, knlDataPage, g_xdata, KCR_RKEK,
CGX_TRIPLE_DES_A, CGX_MAX_SECRET_KEY_LENGTH_B,
                        dpage_5, (secretkey *)argument_5, knlDataPage,
(crypto_cntxt *)NULL,
                        (kcr )argument_2, (publickey *)NULL, TRUE )) !=
CGX_SUCCESS_S)
                  rc = CGX_FAILED_RKEK_S;

      }
      else rc = CGX_FAILED_TOKEN_VERIFY_S;

#if defined(SIM2181) || defined(VSIM2181)
RkekDone:
#endif
      /* ensure the heap is reset */
      pubkey_reset_heap();

      return rc;
}



/*
 ** cgxovlay_save_key
 *
 *  FILENAME: d:\kerntest\src\CGXOVLY1.C
 *
 *  PARAMETERS:
 *
 *  DESCRIPTION:        uncover the black key with the kek, then cover the red
key with the rkek.
 *                              Return the covered red key back to the application
 *
 *    Arg_1 is the secretkey *bk_uncover.       The key to be uncovered.  Cannot
be an LSV
```

```
 *    Arg_2 is the crypto_cntxt *bkek.         The kek required to uncover the
bk.
 *    Arg_3 is the secretkey *bk_returned.     The black key that will be saved
under the rkek and returned to the appl.
 *    Arg_4 is the crypto_cntxt *rkek.         The rkek used to cover the bk.
 *
 *  RETURNS:
 *
 */

UINT16
cgxovlay_save_key(void)
{

      secretkey         sk;                       /* local sk */
      seckey                  kcr_sk;


      if ( (argument_1 == (secretkey *)NULL) || (argument_3 == (secretkey
*)NULL) || (argument_4 == (crypto_cntxt *)NULL) )
              return CGX_FAILED_SAVE_KEY_S;

      /* 1st call the uncover operation, then call the load operation */

      /* uncover the bk_uncovered and place the exposed key to the SCRATCH KCR
*/
      if( (_cgxovlay_uncover_key(KCR_SCRATCH_N, dpage_1, (secretkey
*)argument_1,

dpage_2, (crypto_cntxt *)argument_2) )!=CGX_SUCCESS_S)
             return CGX_FAILED_SAVE_KEY_S;


      /* Next, we must create a secretkey from the seckey that is in the scratch
KCR */
      /* and pass it in the load operation as the 1st argument sk */

      seckey_2secretkey(kcr_get_seckey(KCR_SCRATCH_N, (seckey *)&kcr_sk), &sk);

      /* The uncovered key resides in the scratch KCR */
      /* Next, cover the key under the RKEK by calling cgxovlay_load_key */

      if( (_cgxovlay_load_key((secretkey *)&sk, KCR_SCRATCH_N,
kcr_get_attr(KCR_SCRATCH_N), secretkey_type(&sk), secretkey_length(&sk),
                          dpage_3, (secretkey *)argument_3, dpage_4, (crypto_cntxt
*)argument_4, FALSE) ) !=CGX_SUCCESS_S)
             return CGX_FAILED_SAVE_KEY_S;


      kcr_destroy(KCR_SCRATCH_N);

      return CGX_SUCCESS_S;


}
```